И. О. Леошкевич

ПОЛУЧЕНИЕ АРХИТЕКТУРНО-НЕЗАВИСИМОЙ СЕМАНТИКИ ИСПОЛНЯЕМОГО КОДА

В данной работе под дизассемблированием будем понимать преобразование одной машинной инструкции в представление, отражающее ее семантику, а под дизассемблером — библиотеку, осуществляющую дизассемблирование. Дизассемблеры являются неотъемлемой частью как статических, так и динамических средств анализа исполняемого кода. Целью данной работы является разработка так называемого расширяемого дизассемблера, принимающего на вход описание архитектуры и инструкцию, принадлежащую этой архитектуре, а на выходе дающего формализованное описание работы этой инструкции. Такой подход позволит отделить алгоритмы анализа исполняемого кода от его декодирования, что сделает их реализацию независимой от конкретных архитектур.

Большинство имеющихся в открытом доступе дизассемблеров (LDE, Catchy32, HDE, CADT, libdisasm, Udis86, IDA, OllyDbg и некоторые другие) отвечают лишь за декодирование инструкции. К возвращаемым ими данным относятся длина инструкции, разбиение ее на архитектурно-зависимые поля, а также информация о ее типе и операндах. Использующий их анализатор должен самостоятельно осуществлять переключение по типу инструкции и операндов, что затрудняет или вообще исключает добавление в него поддержки новых архитектур.

LibVEX является частью проекта Valgrind и используется для осуществления бинарной трансляции. LibVEX поддерживает архитектуры x86 и Power. Промежуточным результатом работы LibVEX является набор присваиваний, точно описывающий работу поданной на вход инструкции. В присваиваниях используются выражения, состоящие из констант, обращений к памяти и различных операций, количество видов которых превышает 100. Представление LibVEX можно использовать для полноценной реализации универсальных алгоритмов анализа исполняемого кода. Несмотря на то что большое количество видов операций затрудняет интерпретацию результатов дизассемблирования, анализ зависимостей между операндами с помощью LibVEX реализуется сравнительно легко, чего не может позволить ни один из вышеперечисленных дизассемблеров.

Zynamics BinAudit [1] представляет собой исследовательский проект, направленный на обнаружение уязвимостей в исполняемом коде. Для поддержки различных архитектур в нем используется внутреннее представление REIL, состоящее из 17 микроинструкций, каждая из которых имеет ровно 3 явных операнда и не имеет побочных эффектов. Код для произвольной архитектуры может быть преобразован в REIL и затем передан универсальному алгоритму анализа. Основным отличием представления REIL от LibVEX является замена выражений микроинструкциями. Дизассемблирование и получение представлений как в BinAudit, так и в LibVEX осуществляются специальными функциями, реализованными для каждой поддерживаемой архитектуры.

В статье [2] приводится подход к формализации архитектуры x86 с целью доказательства корректности работы аппаратных механизмов защиты относительно заданной политики безопасности, а также анализа исполняемого кода. Описание архитектуры производится на двух уровнях: на низком уровне декларативно задаются синтаксис и аспекты поведения конкретных инструкций, а на высоком — работа механизмов защиты. Примечательно, что описания составлены на языке LISP — это позволяет использовать их без изменений как данные для анализа с помощью автоматической системы доказательства теорем ACL2, а также как код, являющийся ядром симулятора архитектуры.

TSL [3] (Transformer Specification Language, язык задания преобразований) был разработан для создания описаний архитектур, на основе которых генерируются статические анализаторы на

языке C++. Для описания архитектуры с помощью TSL нужно раздельно описать ее абстрактный синтаксис (правила декодирования) и использующую его конкретную семантику (описания их работы). TSL позволяет создать скелет алгоритма анализа или выбрать существующий и автоматически сгенерировать его реализацию для описанной архитектуры.

Значение алгоритмов исполняемого кода, не зависящих от конкретной архитектуры, трудно переоценить, особенно в свете перехода от наиболее распространенной архитектуры x86 к ее 64-битному варианту. Однако построение их на базе имеющихся в открытом доступе дизассемблеров представляется затруднительным, поскольку они нацелены на конкретные архитектуры. Поэтому наметилась тенденция к реализации многоархитектурных дизассемблеров, особенностями которых являются возможность декларативного описания архитектур и приведения их к общему представлению. В данном разделе описывается подход к построению такого дизассемблера, приводится сравнение с аналогами, рассматриваются его достоинства и недостатки.

Инструкции современных архитектур (x86-64, ARM, Power, SPARC, S/390) можно рассматривать как слова над алфавитом {0, 1}, которые распознаются методом рекурсивного спуска. Это объясняется тем, что более сложные правила разбора потребовали бы усложнения декодирующей компоненты процессора, что нецелесообразно. Таким образом, возможно создавать декларативные описания синтаксиса инструкций для каждой архитектуры и преобразовывать их в эффективные деревья разбора, аналогично тому как это делается утилитами lex и yacc.

Расширяемый дизассемблер включает в себя язык для формального описания архитектур, компилятор для преобразования описаний в деревья разбора и дизассемблер, использующий деревья для получения семантики инструкций. Язык позволяет описывать архитектурные особенности и инструкции пользовательского уровня, в том числе CISC-инструкции, логика которых включает в себя циклы. Цель охватить системный уровень не ставилась. Описания семантики инструкций не имеют побочных эффектов, все аспекты их работы, в том числе и неопределенные, должны указываться явно. Количество допущений, касающихся архитектуры, сведено к минимуму. В частности, типы данных, регистры и кодировки полей инструкций считаются заранее неизвестными. Количество примитивов, в том числе описывающих арифметические операции, также минимально. Рассмотрим составные части языка.

Адресные пространства представляет собой именованные, непересекающиеся, непрерывные участки памяти. Они используются для однородного описания регистров и оперативной памяти. Например, вложенные регистры eax, ax, ah и al в x86 можно описать как части адресного пространства $eax. Регистровые окна SPARC и доступ к нескольким адресным пространствам S/390 также ложатся в эту концепцию.

Типы данных определяют, каким образом следует интерпретировать содержимое участка памяти. Они задаются через операции загрузки и сохранения. Загрузка представляет собой правило для объединения набора битов в целое число, сохранение — правило для разбиения целого числа на набор битов. Единственным встроенным типом является бит, все остальные типы задаются через него. Примерами пользовательских типов являются «байт» и «остроконечное 32-битное слово». Это позволяет анализировать случаи, когда один и тот же участок памяти используется различными способами, например, когда в него записывается знаковое число, которое впоследствии используется как беззнаковое.

В языке используются выражения, представляющие собой целые числа, именованные константы, неопределенные значения, ссылки и операции. Ссылки описывают обращения к памяти с помощью тройки значений: адресного пространства, выражения, описывающего смещение в нем, и типа данных.

Семантика инструкций описывается с помощью высокоуровневых конструкций: присваивания, условного оператора и цикла с предусловием. Клиент расширяемого дизассемблера, однако,

получает в качестве результата дизассемблирования не высокоуровневое представление, а микрокод, который состоит из двух видов операций — присваивания и условного перехода. Ниже приведен фрагмент описания семейства инструкций «bsf» и соответствующего ему микрокода для конкретной инструкции «bsf eax, ebx»:

```
    if(op2 == 0) {                      1: * += (ULE32 [$ebx] == 0) ? 1 :
3
      zf := 1;                         2: [$eflags:6] := 1
       op1 := random((1 << @size) - 1); 3: ULE32 [$eax] :=
random(0xFFFFFFFF)
    } else {                           4: * += 7
      zf := 0;                         5: [$eflags:6] := 0
      bitIndex = Byte [$tmp];
      bitIndex := 0;                   6: Byte [$tmp] := 0
       while(bit(op2, bitIndex) == 0) { 7: * += ((ULE32 [$ebx] >> Byte
[$tmp])
                                              & 1 == 0) ? 3 : 1
        bitIndex := bitIndex + 1;      8: Byte [$tmp] := Byte [$tmp] + 1
      }                                9: * += -2
      op1 := bitIndex;                10: ULE32 [$eax] := Byte [$tmp]
    }                                 11: ...
              Описание                          Микрокод
```

Рис. 1. Фрагмент описания семейства инструкций «bsf» и соответствующего ему микрокода для конкретной инструкции «bsf eax, ebx»

Заметим, что, так как микрокод не является частью языка, правила его генерации могут быть изменены или дополнены без изменения существующих описаний архитектур. Например, можно создавать микрокод, совместимый с Zynamics, или код на языке Си, осуществляющий эмуляцию команд данной архитектуры. Заметим также, что язык и микрокод могут быть расширены для построения на их основе описаний функций операционных систем. В текущем виде их полноценное применение ограничено функциями преобразования данных; для функций, связанных, например, с вводом-выводом, возможно указать лишь затронутые области памяти, что для клиента будет являться достаточной, но крайне пессимистичной информацией.

Синтаксис машинных инструкций описывается с помощью параметризованных правил грамматического разбора. Входными данными являются битовые потоки, позволяющие получить значения, соответствующие следующему биту или заданному пользователем типу данных. Получаемые из битовых потоков значения являются выражениями, что дает возможность сделать их источником не константный массив, а состояние программы, что, в свою очередь, позволяет анализировать некоторые разновидности самомодифицирующегося кода. Результатом применения правил является распознавание данных из входного потока, генерация микрокода, а также изменение переменных. Каждое правило разбора содержит один или несколько вариантов, описывающих распознаваемые им битовые строки. Варианты, в свою очередь, являются последовательностями следующих элементов:

1. *Соответствие константе* (константа = {количество битов}) позволяет проверить равенство следующих битов инструкции заданному числу. Если они равны, то разбор продолжается. В противном случае, если это был первый элемент варианта, то рассматривается следующий вариант, иначе битовая строка считается нераспознанной.

2. *Присваивание переменной* (переменная = выражение | {тип}) либо выражения, либо значения определенного типа, составленного из следующих битов разбираемой инструкции

3. *Соответствие переменной* (переменная = константа) позволяет использовать для распознавания полученную ранее информацию.

4. *Применение правила* (правило(параметр1 -> значение1, параметр2 -> значение2, ...)).

5. *Генерация микрокода* с помощью конструкций высокого уровня.

Возвращаемые расширяемым дизассемблером данные точно описывают результаты работы инструкций. Аналогичным образом работают LibVEX, BinAudit и TSL. Существенным недостатком LibVEX с точки зрения анализа исполняемого кода является большое количество примитивов, что затрудняет интерпретацию результатов. Однако нельзя не отметить, что эта мера упрощает восстановление исполняемого кода из внутреннего представления, что более критично с точки зрения основного применения LibVEX — проекта бинарной трансляции Valgrind. Расширяемый дизассемблер, как и LibVEX, использует выражения для представления семантики инструкций, однако в нем используется меньшее количество примитивов, а также косвенные переходы, что позволяет естественным образом описывать инструкции, логика которых включает в себя циклы. В BinAudit на всех уровнях используется микрокод и отсутствует понятие описания архитектуры. Вместо этого в нем используются исполняемые модули, переводящие исполняемый код в REIL. TSL можно считать наиболее близким по идеологии к расширяемому дизассемблеру, однако имеются отличия. TSL ориентирован на статическое связывание дизассемблера и анализатора, в то время как расширяемый дизассемблер позволяет использовать как статическое, так и динамическое связывание. Расширяемый дизассемблер позволяет пользователю описывать семантику типов данных, что дает анализатору возможность связывать разнородные обращения к одной и той же ячейке памяти, например как к знаковому и беззнаковому числу; в BinAudit и TSL типы данных предопределены и непрозрачны. Единицей адресации в расширяемом дизассемблере является бит, в то время как BinAudit и TSL используют байты.

Производительность текущей реализации расширяемого дизассемблера довольно низка — 100 килобайт в секунду. Это объясняется неоптимальностью реализации битовых потоков, а также объемом возвращаемой информации. Реализация расширяемого дизассемблера на языке Си для платформы x86 вместе с ее описанием занимает 250 килобайт. Расширяемый дизассемблер может работать на любой операционной системе, имеющей виртуальную машину Java. Кроме того, имеется не зависящая от операционной системы сборка для защищенного режима архитектуры x86.

Многие реализации алгоритмов анализа исполняемого кода оперируют понятиями, связанными с инструкциями конкретной архитектуры. Для них характерны следующие два недостатка: невозможность использования без изменений для других архитектур и затруднения при работе с нестандартным использованием инструкций. Поэтому более совершенные системы анализа должны использовать универсальное внутреннее представление — микрокод, в который можно перевести инструкции любой архитектуры. Алгоритмы анализа, в свою очередь, должны быть реализованы в терминах этого представления.

В рамках данной работы был создан язык для формального описания архитектур, покрывающий как их особенности, такие как набор регистров и типы данных, так и синтаксис и семантику инструкций. Был реализован прототип расширяемого дизассемблера [4], использующий описания на этом языке для получения семантики подаваемых на вход инструкций. В данный момент в него входят описание подмножеств архитектур x86 и S/390 и некоторых функций ОС Windows, компилятор для описаний архитектур, а также непосредственно дизассемблер на языках C и Java. Весь программный комплекс распространяется по лицензии BSD.

Следующие аспекты не были включены в архитектуру расширяемого дизассемблера, но заслуживают упоминания как потенциальные направления для улучшения, некритичные, однако, для предполагаемых на данный момент областей применения: операции с плавающей

точкой, ввод-вывод, страничное преобразование, защита, кэш, многопроцессорные системы и прерывания. Расширяемый дизассемблер является базой для построения алгоритмов анализа исполняемого кода. В данный момент на его основе ведется разработка алгоритма, совмещающего абстрактную интерпретацию и символьное выполнение исполняемого кода с целью поиска в нем потенциально уязвимых участков, в частности ошибок переполнений буфера. Помимо этого видятся перспективными такие применения, как построение справочных материалов на основе описаний архитектур, оптимизация произвольного «мусорного» кода с помощью систем компьютерной алгебры, автоматизированный разбор наборов инструкций виртуальных машин, а также идентификация структур данных и анализ их свойств — от простых стековых переменных до связанных структур в динамической памяти.

СПИСОК ЛИТЕРАТУРЫ:

1. *Dullien T., Porst S.* Platform-independent static binary code analysis using a meta-assembly language // CanSecWest, 2009.
2. *Sandip R.* Towards a formalization of the x86 instruction set architecture. Technical Report TR-08-15, 2008.
3. *Lim J., Reps T.* A system for generating static analyzers for machine instructions // CC '08: International Conference on Compiler Construction, 2008.
4. *Leoshkevich I.* Q disassembler. URL: http://sourceforge.net/projects/q-d/.